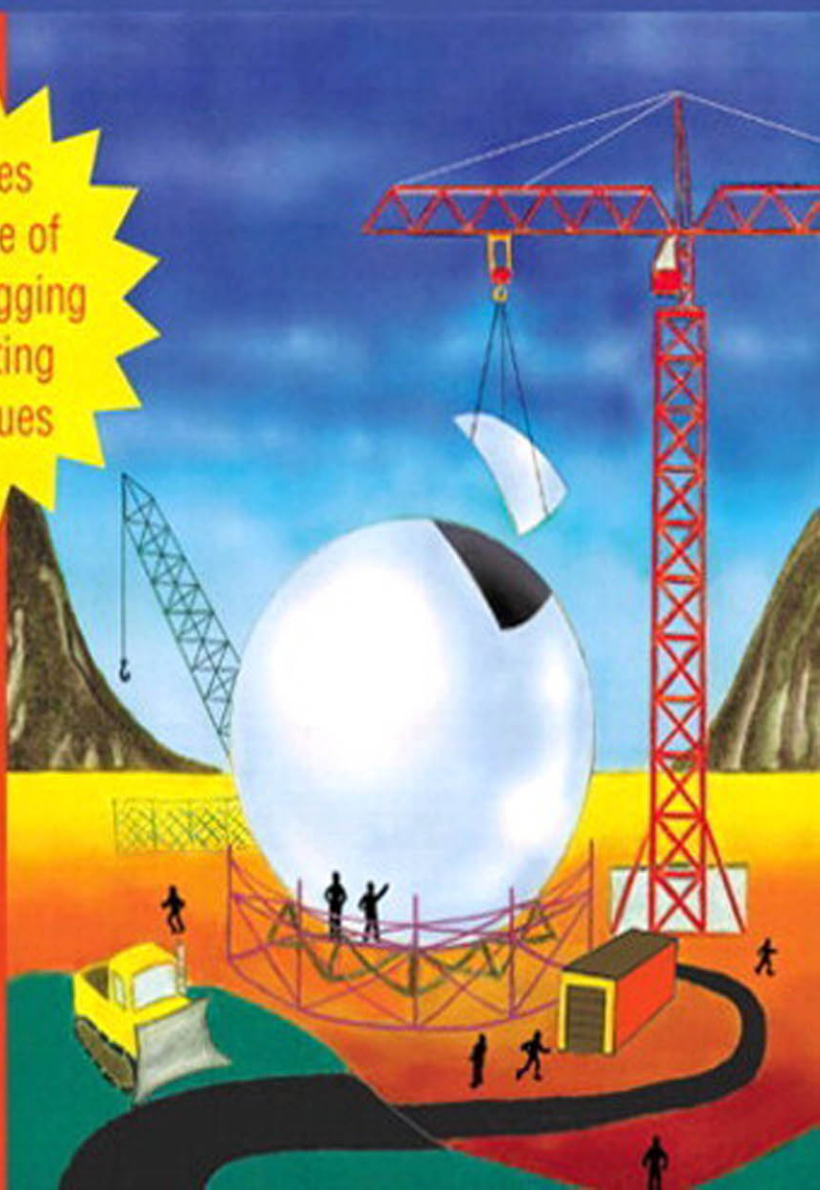
Perl Debugged

Includes coverage of CGI debugging and testing techniques



Peter Scott
Ed Wright

Perl Debugged

Associativity tells you what order to evaluate multiple operators of the same precedence appearing together in an expression. So the arithmetic expression `3 - 4 + 5` is evaluated as `(3 - 4) + 5` because while - and + have the same precedence, they are left-associative, meaning that left-most elements take precedence over right-most elements. "Nonassoc" means that the operators don't associate; in other words, you can't have more than one in a row without parentheses to create a valid meaning. So while some languages let you write[5]

```
if (0 <= $percent <= 100)
```

Perl isn't one of them.[6]

As the `perlop` manual page says, "Operators borrowed from C keep the same precedence relationship with each other, even where C's precedence is slightly screwy." It turns out that some of the precedence rules for C were chosen to match those of its predecessor B, even where *those* were slightly screwy. (Backward compatibility has its pitfalls.)

So most of the precedence gotchas in C are inherited by Perl. For instance, if you're checking a variable that contains a selection of flags logically ORed together against a candidate (as we will do in Chapter 5), the line

```
warn "Database opened\n" if ($opt_D & $DB_LOG != 0)
```

has an unfortunate flaw, because it is evaluated as

```
warn "Database opened\n" if ($opt_D & ($DB_LOG != 0))
```

So if, for instance, `$opt_D` was `6`, and `$DB_LOG` was `2`, the `warn` statement would be incorrectly omitted since `2 != 0` evaluates to `1`, and `1 & 6` is `0`, which is false.

---

5. One of the first languages Peter learned was BCPL, which allowed this.
6. Yet. It's been mooted for Perl 6.

The cure lies in simplicity:

```
warn "Database opened\n" if ($opt_D & $DB_LOG)
```

at the cost of infinitesimal clarity.

Here's another example of precedence combined with optional parentheses around function arguments getting us into trouble:

```
vec $v, $offset, 32 = 1;   # Wrong
```

The programmer who thought that since the prototype for `vec` is `$$$` that the processing of its arguments would terminate after `32` failed to realize that = has a higher precedence than the commas separating list arguments. Therefore Perl parses it as:

```
vec($v, $offset, (32 = 1));
```

and emits the error

```
Can't modify constant item in scalar assignment.
```

What other precedence pitfalls pursue Perl programmers? A few follow.

4.2.1 Regex Binding

Let's say that you want to check whether the next element of an array matches a regular expression, and you write

```
if (shift @marsupials =~
          /platypus|wallaby|(kanga)?roo|joey/) ...
```

This results in a warning (with `-w`) and an error:

```
Applying pattern match to @array will act on scalar(@array)..
Type of arg 1 to shift must be array (not pattern match)...
```

Reason: the binding operator =~ is above "named unary operators" in Table 4-1. So it tries to bind the regex to `@marsupials` (which must put the array in scalar context, hence the warning), and then shift the result of the regex match (which results in the error).

Fortunately, there are so many ways of coding this right that this is an unlikely pitfall to encounter by accident. Like all precedence misunderstandings, inserting parentheses is one way to fix it:

```
if (shift(@marsupials) =~
          /platypus|wallaby|(kanga)?roo|joey/)
```

Use parentheses when in doubt about precedence; they won't hurt.

4.2.2 Arithmetic on `keys`

Suppose you have a hash `%h` and you want to find out how many elements are in it, so you type:

```
print "Number of elements: " . keys %h . "\n";
```

reasoning with impeccable logic that the dots will force `keys` into scalar context and therefore return the number of keys in `%h` instead of the list of them. However, Perl responds with

```
Type of arg 1 to keys must be hash (not concatenation)
```

A quick glance at Table 4-1 shows us that "named unary operators" rank below the `.` operator in precedence; therefore Perl parses the statement as

```
print "Number of elements: " . keys (%h."\n");
```

which, unsurprisingly, makes no more sense to Perl than it does to us. We can fix it easily enough by leaving the first dot in and turning the second one into a comma:

```
print "Number of elements: " . keys %h, "\n";
```

(This comma is evaluated as the low-precedence "list operators (rightward)" element from Table 4-1.) We can generalize this to all the other named unary operators (functions prototyped to take a single argument), of course. Peruse the `perlfunc` manual page to find out what those are.

4.3 Regular Expressions

You may be accustomed to using `s//A New Beginning/` to insert some text at the beginning of the current line in some applications. In Perl (as in `sed`), the empty regular expression doesn't match a zero-length substring; it matches whatever the previous regular expression matched. If there isn't a previous regular expression, Perl and `sed` part company: Perl treats it as matching the beginning of the string; `sed` says it's an error.

4.4 Miscellaneous

4.4.1 Autovivification

You wouldn't ordinarily expect to alter the value of something just by looking at it. So when that happens (even if for very good reasons), it's worth drawing attention to it so you can look out for it.

Autovivification is an expensive Perl term that you can impress your friends with. It means that certain variables get created for you if necessary. Perl has extremely friendly syntax for creating multidimensional data structures. For instance, let's say you're populating the hash `%state_info` from section 4.1.3; you have first-level keys that are state abbreviations and second-level keys that are names of state attributes, one of which names the largest cities. You can insert the entry for those cities right off the bat:

```
$state_info{NY}{Megalopoles} = ['New York',
                                'Albany',
                                'Rochester'];
```

What's remarkable about this is how easy it is. A C programmer getting to know Perl would look at this and say, "Er, don't you need to at least insert a reference to a hash in the first-level value before this...?" What they mean is, the preceding code reveals that the values of `%state_info` are references to hashes; if `$state_info{NY}` doesn't exist yet, then why doesn't the assignment above blow up when it tries to dereference an undefined value to find where the value for the second-level key `Megalopoles` is? To do the same in C, a programmer would first have to `malloc` a new `state_info` struct with a field of `NY`, insert it into a binary tree, `malloc` another struct with a field of `Megalopoles`, insert a pointer to it in the previous struct, then `malloc` three more structs with the city names, link them into a list, and put a pointer to the head of the list in the `Megalopoles` struct. Pant, pant.

That Perl doesn't blow up is another example of its philosophy of, "Don't throw an exception if you can do something useful." When the assignment above is executed, Perl will look to see if there is a key `NY` in the `%state_info` hash; if there isn't (or if there is but the corresponding value is `undef`), it will create one with a value containing a reference to an anonymous hash with a key `Megalopoles` and a value of a reference to an anonymous array containing the three listed cities. Now you know what autovivification is.

What does this have to do with the Heisenbergian lead-in to this section? Well, sometimes when you're just looking to see if a second- (or third-, etc.) level hash key exists, Perl can't help but create a first-level value through this autovivification process. For instance,

```
my %state_info;   # So you know it's empty at this point
print "The Big Apple is here!\n"
    if exists $state_info{NY}{Megalopoles};
use Data::Dumper;
print Dumper \%state_info;
```

prints

```
$VAR1 = {
          'NY' => {}
        };
```

In other words, just looking to see whether the key `Megalopoles` existed caused Perl to autovivify the intermediate hash. The documentation says that this behavior may be fixed in a future release of Perl, but it also appears to be extremely hard to do so.

Before testing the existence of a lower-level hash key, test the existence of the higher-level keys if there's a chance they may be absent.

We'll talk more about the useful `Data::Dumper` module in Chapter 5.

4.4.2 `split`

The `split` function deserves its own section. It has almost enough special cases to qualify as its own programming language. Our first potential gotcha emerges when we're validating a Unix `passwd` file for the correct number of fields on each line:

```
@ARGV = '/etc/passwd';
while (<>)
   {
   print "Wrong # of fields in line $.\n"
      unless split /:/ == 7;
   }
```

Our first problem is that this doesn't find any of the lines that have the wrong number of fields; it silently fails. We've just created another precedence problem: referring back to Table 4-1, we see that list operators have a lower precedence than the == operator, and `split` is a list operator; therefore we've just given it a first argument of `/:/ == 7`.